Implementing Distributed Controllers for Systems with Priorities

Imene Ben-Hafaiedh Susanne Graf Hammadi Khairallah

Universite Joseph Fourier/VERIMAG
Grenoble, France

Tunisia Polytechnic School
Tunis, Tunisia

benhfaie@imag.fr graf@imag.fr khairall@imag.fr

Implementing a component-based system in a distributed way so that it ensures some global constraints is a challenging problem. We consider here abstract specifications consisting of a composition of components and a controller given in the form of a set of interactions and a priority order amongst them. In the context of distributed systems, such a controller must be executed in a distributed fashion while still respecting the global constraints imposed by interactions and priorities. We present in this paper an implementation of an algorithm that allows a distributed execution of systems with (binary) interactions and priorities. We also present a comprehensive simulation analysis that shows how sensitive to changes our algorithm is, in particular changes related to the degree of conflict in the system.

1 Introduction

A distributed system is a collection of components, or processes, communicating by explicit message passing. These components are intrinsically concurrent and knowledge about their respective states can be obtained only through communication. Thus determining the exact global state of such systems is not a trivial task [6], and in general not required. The motivation of this work is to generate a distributed implementation of systems defined as a set of synchronizing processes and a set of *priority* constraints among these synchronizations or interactions.

Specifying priorities amongst a set of alternative system interactions is interesting in different contexts. For example, it is likely that amongst a set of enabled synchronizations amongst subsets of components, one will prefer those involving larger subsets. Another typical example of the use of priorities are processes which for different activities require one or more resources amongst a shared pool of resources.

There exist several abstract frameworks allowing to represent specifications with priorities, such as process algebras with priorities or prioritized Petrinets. We present here our results for a simple formalism, motivated by the application to BIP [7, 3]. In [4], we have proposed an algorithm defining a controller for each of the processes and that allows a distributed execution of such prioritized specification.

Here, we discuss an implementation of this algorithm and an evaluation of different performance metrics. In a number of experiments, we measure the *execution-time* and *message-count* of the algorithm as a result of variations in different model parameters, in particular, variations in the degree of *conflict* of the system. We also compare our algorithm to α-core algorithm based on experimental results provided in [10].

The paper is organized as follows: Section 2 introduces the specification formalism and the relevant notions of concurrency and conflict for discussing the correctness of a distributed implementation, and then a description of the problem tackled by our algorithm. Section 3 gives a brief description of the algorithm and how it works on an illustrative example.

M.R. Mousavi and G. Salaün (Ed.): International Workshop
on the Foundations of Coordination Languages and
Software Architectures (FOCLASA 2010).
EPTCS 30, 2010, pp. 31–46, doi:10.4204/EPTCS.30.3

The main contribution of this paper is given in Section 4 which describes the implementation of the algorithm and our experimental results.

2 Distributed Controllers for Systems with Priorities

2.1 Processes interacting through synchronizations and priorities

As a formalism for describing systems with synchronizations and priorities — which we then want to execute in a distributed fashion — we choose a very basic formalism for describing systems consisting of a set of components or processes which interact through n-ary synchronizations, and where there may be given a partial order on the set of synchronizations defining a priority amongst them. The motivation for this work roots in the need for algorithms for distributed executions of BIP models [7, 3, 5] or for prioritized Petrinets. Indeed, there are straightforward mappings between the simple formalism considered here and the before mentioned more evolved ones. We consider processes to be represented by labeled transition systems where labels represent a set of interactions on which several processes synchronize. That is, our formalism is close to Petrinets where interactions correspond to (joint) transitions. On the other hand, in BIP we would associate globally unique port names with components and identify an interaction by a set of ports of different components. In terms of reusability BIP components are preferable, but here our aim is to be able to define the problem to be solved and the algorithm solving it in a most intuitive manner. And for this purpose, naming interactions is most appropriate.

Definition 1 (Process). *A process is a* Labeled Transition System *(LTS) represented by a tuple* $(Q, q^0, \mathscr{P}, \delta)$ *where* Q *is a set of states,* $q^0 \in Q$ *is an initial state,* $\mathscr{P}$ *is a set of labels representing interactions and* $\delta \subseteq Q \times \mathscr{P} \times Q$ *is a transition relation.*

As usually, we write $q_1 \xrightarrow{a} q_2$ instead of $(q_1, a, q_2) \in \delta$ and $q_1 \xrightarrow{a}$ instead of $\exists q' \in Q, q \xrightarrow{a} q'$. We also write sometimes $q_1 \xrightarrow{\varepsilon} q_2$ when $q_2 = q_1$.

Composed systems Given a set of n processes $K_i = (Q_i, q_i^0, \mathscr{P}_i, \delta_i)$ for $i \in [1,n]$, its composition is a process defined on the set of interactions $\mathscr{P} = \bigcup_{i=1}^n \mathscr{P}_i$[1].

Definition 2 (Interleaving semantics of composition). *The composition of n processes* K_i

is denoted $K = \|(K_1, \dots, K_n)$ *and its semantics is defined by the LTS* $(Q, q^0, \mathscr{P}, \delta)$ *where* $Q = \prod_{i=1}^n Q_i$, $q^0 = (q_1^0, \dots, q_n^0)$, $\mathscr{P} = \bigcup_{i=1}^n \mathscr{P}_i$. *Now the transition relation* $\delta \subseteq Q \times \mathscr{P} \times Q$ *is defined for each* $a \in \mathscr{P}$ *as the smallest subset of transitions obtained by applying the following rule, where I is the set of indexes of the processes having a in their alphabet:*

$$\frac{\forall i \in I.\ q_i^1 \xrightarrow{a} q_i^2 \wedge \forall i \notin I.\ q_i^1 = q_i^2}{(q_1^1, \dots, q_n^1) \xrightarrow{a} (q_1^2, \dots, q_n^2)}$$

This means that a transition from state q in the composed system that $\|(K_1, \dots, K_n)$ consists for any $a \in \mathscr{P}$ of the joint execution of an a-transition in all the processes K_i having a in their alphabet. That is the fact that an a-interaction can be fired is defined locally in the substate of the processes involved in this interaction.

[1] in fact, we choose a strict subset which means that decide to block a set of interactions offered by some of the components and we can rename some interactions to τ which means making them local; but this is not important for our algorithm.

Priorities A (partial) priority order amongst interaction may restrict the choice of interactions that can be fired in a given state. That is, priorities may restrict nondeterminism. The semantics of priorities is global, that is, in presence of priority, whether a transition is enabled may depend on the entire global state.

Definition 3 (Priority order). *A* priority order *denoted by $<$ is a strict partial order on a set of interactions. We denote that an interaction a has lower priority than b by $a < b$.*

Definition 4 (System controlled by a priority order). *The semantics of a system $S = (Q, q^0, \mathscr{P}, \delta)$* controlled *by a priority order $<$ defines an LTS $(Q, q^0, \mathscr{P}, \delta_<)$ where $\delta_<$ is defined by the following rule, where we denote by I the indexes of the processes involved in a and by q_I the substate of q defined by I:*

$$\frac{q_I \xrightarrow{a}_S q'_I \ \wedge\ \nexists b \in \mathscr{P}.\ (a < b \wedge\ q \xrightarrow{b})}{q \xrightarrow{a}_< q'}$$

Thus, only interactions that are locally enabled in all concerned components, and furthermore not inhibited by an interaction with higher priority, may be fired. An interesting property of priorities is the well-known fact that they allow restricting the behavior of a system by guaranteeing that no new deadlocks are introduced by this restriction. This is the reason why we want to use priorities to "control" systems. We denote the resulting controlled system by $(S, <)$.

We now introduce notations allowing to distinguish between the enabledness of a transition locally in some process, in the uncontrolled system S and in the controlled system $(S, <)$, where we suppose in the following S to be defined as the composition of $P_i = (Q_i, q_i^0, \mathscr{P}_i, \delta_i)$ with $i \in [1, n]$ and $\mathscr{P} = \cup \mathscr{P}_i$.

Definition 5 (locally ready, globally ready, enabled interaction). *Consider a global state $q \in Q_S$ such that $q = (q_1, \dots, q_n)$ and an interaction $a \in \mathscr{P}$.*

- *For i such that $a \in \mathscr{P}_i$, a is* locally ready *in q_i iff $\exists\ q'_i \in Q_i,\ s.t.\ q_i \xrightarrow{a}_i q'_i$*
- *a is* globally ready *in q iff $\exists q' \in Q_S.\ q \xrightarrow{a}_S q'$*
- *a is* enabled *in q iff a is* globally ready *in q and no interaction with higher priority is also globally ready in q, that is, iff $q \xrightarrow{a}_{(S,<)}$.*

Note that only enabledness is related to priorities, and enabledness of a implies global readiness which in turn implies local readiness in all processes which have a in their alphabet. We are interested in distributed executions of $(S, <)$. We therefore define a notion of *concurrency* and *conflict* of interactions, such that in a distributed setting we may allow the independent execution of concurrent interactions (so as to avoid global sequencing). We distinguish explicitly between the usual notion of conflict which we call structural conflict, and a conflict due to priorities.

Definition 6 (Concurrent and conflicting interactions). *Let a, b be interactions of $\mathscr{P}$ and $q \in Q$ a global state in which a and b are globally ready.*

- *a and b are called* concurrent *in q iff a and b are globally ready in q and the set of processes I_a, I_b involved in a, resp. b are disjoint.*

  That is, when a is executed then b is still globally ready afterwards, and vice versa, and if executed, both execution sequences lead to the same global state.
- *a and b are called* in structural conflict *in q iff they are not concurrent in q, that is a and b are alternatives disabling each other.*
- *a and b are in* prioritized conflict *in q iff a and b are concurrent in q but $a < b$ or $b < a$ holds.*

Note that in case of prioritized conflict, it is known which interaction cannot be executed, whereas in case of structural conflict, the situation is symmetric. Note that there is a particular situation, called a prioritized confusion, in which a and b are concurrent and both of maximal priority in q, but when a is executed a state q' is reached in which b is still globally ready but not anymore of maximal priority. Such a situation can be statically detected and eliminated by additional priorities. We consider only specifications without this kind of confusion.

2.2 Problem description

A system $(S,<)$ is defined by a *composed system* S — of set of processes P_i, a set of interactions and a priority order $<$ to be enforced, our goal is to define a distributed implementation for $(S,<)$. We define an algorithm which constructs such a distributed implementation by defining for each process a *local controller* such that the joint execution of all processes P_i and their corresponding controllers guarantee the following:

1. Any sequentialisation of an execution of the obtained concurrent execution is an execution of $(S,<)$, that is executions of S respecting $<$
2. if $(S,<)$ is deadlock free, then no execution will deadlock

Sequentialisations are obtained by arbitrarily ordering *concurrently* executed interactions. Controllers are described as protocols interacting amongst each other by messages. Our aim is for each process P_i to be able to execute a next transition as quickly as possible, and not to minimise the number of messages sent.

3 The protocol

In this section, we provide here a description of the protocol used by each *local* controller associated to each process in a system $(S,<)$, which can be found in [4]. Each system has a fixed number of processes P_i. We consider here only binary interactions.

The protocol performs communication by messages exchanged between processes so as to be able to decide about a next interaction to be fired jointly. We also illustrate how our protocol behaves on a simple example. We also assume that the internal activities of processes are terminating. As quite usually, we assume that the message passing mechanism ensures the following basic properties: 1) any message is received at the destination within a finite delay; 2) messages sent from location L_1 to L_2 are received in the order in which they have been sent; 3) there is no duplication nor spontaneous creation of messages.

3.1 Description of the protocol

We now describe the controllers of individual processes which enforce correct executions, that is joint executions of synchronizations and adherence to the global priority order. It is understood that what we call in the following "process" is in fact a *controlled process* obtained by executing the process and its controller in the context of all peer processes.

For each interaction a involved in at least one priority rule, one of the involved processes P_i plays the role of the negotiator for a. If there exists at least one interaction with higher priority, the role of the negotiator is to check for the enabledness of a, and if there exists at least one interaction with lower priority, its role is to answer readiness requests.

The Controller associated with each process, maintains a set of data structures shared and maintained by the different subtasks of the controller: *readySet* (resp. *enabledSet*) contains the set of interactions

which are known to be globally ready (resp. enabled) in the current local state q and *possibleSet* maintains the set of interactions that are locally ready. Note that *possibleSet* contains purely local information which can be calculated immediately when entering a new local state. The other two sets are calculated by a series of message exchanges, and the complete information is generally not calculated but as soon as an interaction is known to be enabled, its triggering will be initiated.

The general structure of the controller for each individual process P_i is shown in Figure 1. The overall controller — and the process to be controlled — are represented as a set of parallel activities which we call threads, and which in our implementation are realized by Java threads (see Section 4) with a shared memory space and a set of shared message buffers. The detailed algorithms of these threads are given in an appendix at the end of the paper.

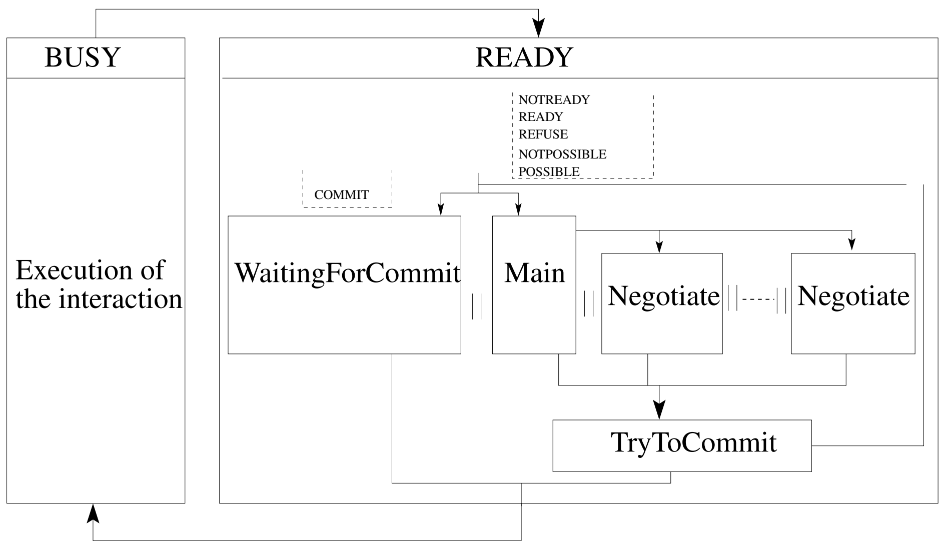

Figure 1: Structure of the protocol for one process

Indeed, incoming messages are stored until one of the activities is ready to handle them. We use several FIFO buffers which are chosen such that the order amongst messages stored in different buffers does not influence the algorithm; in particular, they are used by concurrent threads. A buffer, which is read only by the thread *Main*, stores messages of the form $POSSIBLE(a)$, $NOTPOSSIBLE(a)$, $READY(a)$, $NOTREADY(a)$, and $REFUSE(a)$. A second buffer stores messages of the form $COMMIT(a)$, this buffer is read first by thread *WaitingForCommit*, then by $TryToCommit$.

The role of each message is described in Table 1. Given that we are handling binary interactions, we do not explicitly mention the recipient nor the sender.

The controller C_i associated with P_i is either in state *Ready* or in state *Busy*. In state *Busy*, P_i executes the local action corresponding to the interaction chosen by the protocol. Incoming messages are stored and will not be handled until the controller moves into state *Ready*. In state *Ready*, C_i looks for a next interaction to fire, proceeding as follows:

- The $Main$ thread starts by checking its locally ready interactions (*possibleSet*) for interactions that are globally ready (see Algorithm 1). To check the global readiness of an interaction a, messages of the form $POSSIBLE(a)$ are exchanged, and peers in which a is currently not locally enabled respond with $NOTPOSSIBLE(a)$ after which the requesting controller "abandons" a until the process changes its state or the peer enters a state in which a is locally enabled and sends a $POSSIBLE(a)$.

Message	Description
POSSIBLE	Offer an interaction (which is locally ready)
NOTPOSSIBLE	respond that an interaction is not locally ready
READY	Ask about the global readiness of an interaction
NOTREADY	Respond that an interaction is not globally ready
COMMIT	Commit to an interaction (cannot be undone by P_i)
REFUSE	Inform that a process cannot commit to an interaction

Table 1: Messages used by the algorithm

Whenever it is detected that an interaction a for which it plays the role of a negotiator is globally ready, a thread $Negotiate(a)$ is created which checks whether a is enabled. If an interaction with maximal priority is globally ready, it is immediately known to be enabled and a COMMIT phase is entered (see further down).

- the $Negotiate(a)$ thread checks the enabledness of an interaction a (see Algorithm 2). By sending a $READY(b)$ message to all negotiators of interactions b with higher priority than a, it checks whether their interactions are globally ready (and thus a cannot be executed now).

  In turn the negotiators of b, as soon as they are not $BUSY$ and have found out whether b is globally ready, respond positively or negatively as soon as they have the information available[2].

- The $Main$ thread handles local priorities locally. Whenever an interaction b is known to be globally ready, it kills all threads $Negotiate(a)$ if $a < b$.

- Concurrently to $Main$, the thread *WaitingForCommit* handles incoming *COMMIT* messages (see Algorithm 3). Whenever a $COMMIT(a)$ is received — which means that a is enabled and that the local process should commit to it[3] — all other negotiation activities are terminated and a response $COMMIT(a)$ is sent back to the peer.

- As long as no commit phase is initiated by a peer, $Main$ tries to commit to the first interaction found enabled (as a way to handle local conflicts) by activating $TryToCommit$. *WaitingForCommit* terminates once $TryToCommit$ is activated, to avoid multiple commits in the same state of P_i.

- $TryToCommit(a)$ sends a $COMMIT(a)$ message to the corresponding peer and waits for a response (see Algorithm 4). Note that if $TryToCommit$ fails committing to a because it receives a $REFUSE$ message — in that case the peer has committed to a conflicting interaction — the process starts again by checking the global readiness of its locally ready interactions. Indeed, as the peer has committed to another action its state may have changed. For the interactions a for which there exists at least one interaction with higher priority, the commit procedure is always initiated by the negotiator of a who is the first one to know about a's enabledness.

- Finally, the thread $AnswerNegotiators$ is always active if the process P_i is the negotiator for at least one interaction a that dominates some other interaction. This thread receives messages of the form $READY(a)$. It returns $NOTREADY(a)$ if a is in the *notReadySet*, and otherwise defers the answer until the status of a is known.

[2] In fact, it is sufficient that $NONREADY(b)$ messages are sent as a is blocked anyway as long as it does not have a response concerning b.

[3] the existence of the thread *WaitingForCommit* means that no other actions is in its commit phase yet

3.2 Decision cycles

In order to avoid deadlocks due to decision cycles amongst interactions in conflict, we introduce a notion of *cycle* representing potential decision cycles. We denote by *inter(a,*P_1*,*P_2*)* the fact that interaction a involves processes P_1 and P_2. A *cycle*, C_A is a set of interactions $A = \{a_i\}_{i=1}^{n}$ for which the following holds: there exist n processes $\{P_i\}_{i=1}^{n}$, such that $\bigwedge_{i=1}^{n} inter(a_i,P_i,P_{i+1\,mod\,n})$. In addition, we require that there exists at least one global state in which all conflicting interactions are enabled. A *cycle* C_A bears indeed a risk of deadlock or livelock in a state in which all interactions of C_A are enabled. Indeed, it represents a symmetric situation for all involved processes, where each process could wait forever a response to a *COMMIT* (deadlock) or propose a choice of a next interaction representing a differnet solution than the one chosen (locally) by all others, reject it by sending a *REFUSE* and then start all over again forever (livelock). This is a well-known problem in the context of communicating processes. In [1] a total order over the system interactions is defined, which allows to avoid deadlock by executing the interaction with higher order if an actual conflict occurs. In [10], a similar solution is proposed by imposing a total order over all processes, which breaks the cycle by always executing the interaction proposed by the process with higher order.

The solution we propose is to detect statically the set of (minimal) cycles of the system. Then, in a second step, we define for each cycle statically a process of the cycle playing the role of the *Cyclebreaker*. This particular process will arbitrate when a blocking situation actually occurs. This approach avoids defining a total order of all interactions or processes which is useless if there is no cycle.

Illustrative example Figure 2 depicts an example representing a cycle. The system consists of 4 components: 3 processes $\{P_1,P_2,P_3\}$ forming a cycle C_A for the set of interactions $A = \{a,b,c\}$, and a completely independent process P_4. The existence of a cycle can be concluded from the structure and the behaviors of the processes (the interactions a, b, c are always enabled and in conflict). If no priority rules are defined on the set of interactions A, then the algorithm — as explained so far – may end in a deadlock. A possible deadlock scenario is depicted on the right side of Figure 2. This occurs when P_i sends a *COMMIT* message to P_{i+1} and waits for it. Which means that each P_i is waiting for a response from its peer who has made another choice and is waiting as well. According to the proposed solution, let us suppose that P_2 is chosen as the *Cyclebreaker* of C_A. According to Algorithm 4 (as described in Figure 2), whenever process P_i which is already engaged in committing an interaction and which receives a *COMMIT* for a different interaction, will send back a *REFUSE* message only if the *COMMIT* comes from a process which is not the *Cyclebreaker*. This breaks the cycle. Independently, the process P_4 can perform whenever it is possible the interaction d. We have proven the correctness of this algorithm in [4]. In this paper, we present an implementation and its use in a number of experiments.

4 Implementation and experimental results

We have implemented the protocol described in Section 3 using Java 1.6 and Message Passing Interfaces (MPI) in order to experiment its efficiency on examples of different nature. We have used the MPI library [11] to perform the communication layer of our algorithm because of its good performance, usage facility and its portability [8]. In our prototype, the exchange of messages between processes is performed at the MPI layer and all the computation operations of our algorithm are performed at the Java program level (see Figure 3). In this section, we show how we have evaluated the performance of our algorithm on hand of the implemented prototype. Tests have been run on a set of 2.2 GHz Intel machines with 2 GB RAM, in a configuration where each physical machine hosts only one process. We have however

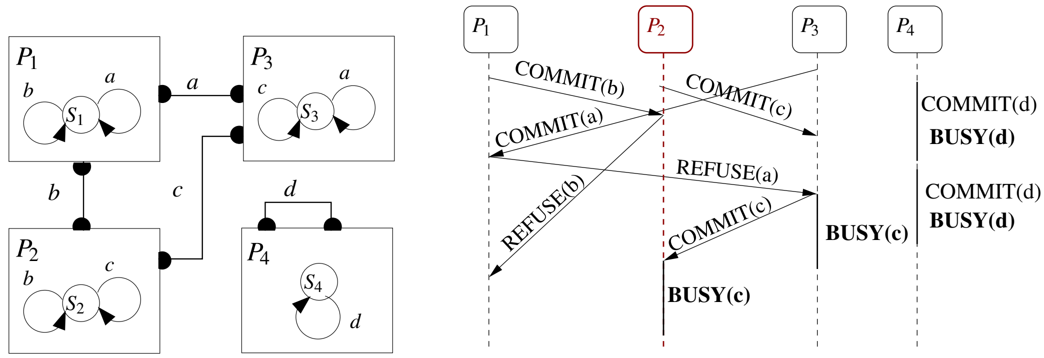

Figure 2: An example with cycle and independence

not made sure that no other application is running during the experiments, just that the overall charge on each machine is "low".

Our experiments evaluated essentially two metrics which are comparable to those used also in [10]: the first is a metric called *message-count* which measures the (average) number of messages required to schedule an interaction for execution, starting from the moment on that it is ready in one of the involved processes. The second one is called *Response-Time* and is defined as the sum of two other metrics *Sync-Time* and *selection-Time*: *Sync-Time* measures the (mean) time taken by the algorithm to ensure that a given interaction is globally *ready*, again starting from the moment where it is locally ready in at least one of the peers[4]. *Selection-Time* measures the (mean) time taken by the algorithm to select an interaction for execution once it has been found enabled.

All metrics are measured for a given system by experimenting with different choices of parameters. We then analyze how variations of parameters affect the considered metrics and compare them to theoretical analysis on the algorithm.

We also compare for an example without priorities the *message-count* metric obtained for our algorithm and for an implementation of the α-core algorithm. We could not compare execution times because the implementation of α-core we have at hand cannot be run in the same setting and the data provided in [10] are obtained in a incomparable setting as well.

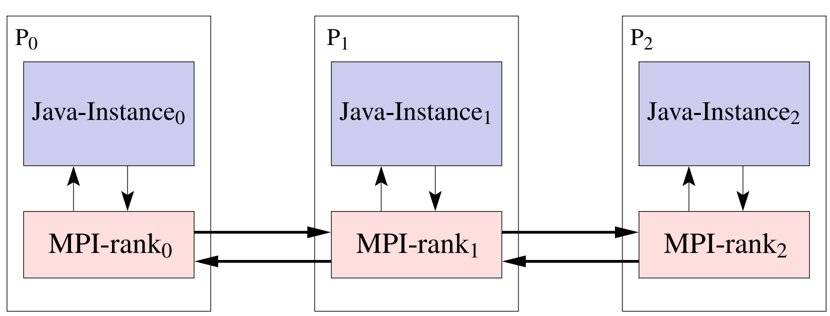

Figure 3: Implementation layers

[4]an alternative option would be to measure only from the moment on where the interaction is already enabled, that is only the time required to "detect" this enabledness; this is however quite difficult to evaluate in a distributed setting.

4.1 Sensitivity to the degree of conflict

First, we study the sensitivity of our algorithm to the degree of conflict in a given system. The degree of conflict (d) is measured by the number of interactions that may be in actual conflict with any (or a particular) interaction. Remember that we distinguish between *structural* and *prioritized* conflict (see Definition 6).

4.1.1 Sensitivity to prioritized conflicts

The purpose of our algorithm is to ensure correct synchronization between processes by respecting global priorities. We first show some results concerning *prioritized* conflicts.

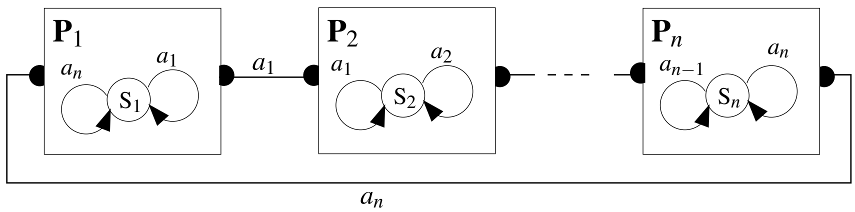

Figure 4: System pattern for experiments

An evaluation has been undertaken using the example depicted in Figure 4 with a single global state. For each considered configration, the system has been executed several times, and each execution has been terminated at the execution of the first interaction. The system consists of a set of n processes, and a set of n interactions building a circular chain. This pattern is flexible and it allows as to observe how our algorithm performs in different situations. In fact, we can easily add both local and global priorities.

Considering a given system, that is a composition $\|(P_1,...,P_n)$, d can be increased by adding priority constraints. Here, we simply count the maximal number of priorities in which a single process is involved, in order to obtain the degree d, but as the discussion will reveal, finer measures could also be considered. interactions of different processes of S than $<_2$.

As already explained, our experiments are performed on a system as depicted in Figure 4, for $n = 4$ and using the following priorities to achieve different degrees of conflict, where process P_2 — which is chosen as the negotiator of a_1 — is the process which in all cases is involved in all the priorities, whereas other processes are involved in at most two of them: $d = 0$: no priorities, $d = 1$: $a_2 < a_1$, $d = 2$: $a_2 < a_1 \wedge a_3 < a_1$, $d = 3$: $a_2 < a_1 \wedge a_3 < a_1 \wedge a_4 < a_1$, $d = 4$: $a_2 < a_1 \wedge a_3 < a_1 \wedge a_4 < a_1 \wedge a_3 < a_2$. We have measured the average *message-count*, *response-time*, *sync-time* and the *selection-time* for all for cases.

Variation of metric message-count Figure 5 shows that — as expected — the number of messages exchanged in order to execute the first (and unique) interaction increases with the degree d of the system. Increasing d means that more interactions are involved in priority rules, and thus more messages of type *READY* are exchanged, and globally less interactions can be executed. In the case chosen for $d = 1$, the priority is defined by the rule $a_2 < a_1$ which is a *local* priority involving only process P_2. Thus, no negotiations and no *READY* messages are needed which makes in principle, the same *message-count* as for $d = 0$. This is confirmed by Figure 5 showing a non significant difference between $d = 0$ and $d = 1$. For the case $d = 2$, the selected priorities are $a_2 < a_1$ and $a_3 < a_1$. This means that the negotiator of a_3 (P_3) has to send a *READY* message to the negotiator of a_1 (P_2) and the latter has to send back as a response a *READY* message which makes 2 extra messages added comparing to the case of $d = 0$. This is confirmed by the experimental results.

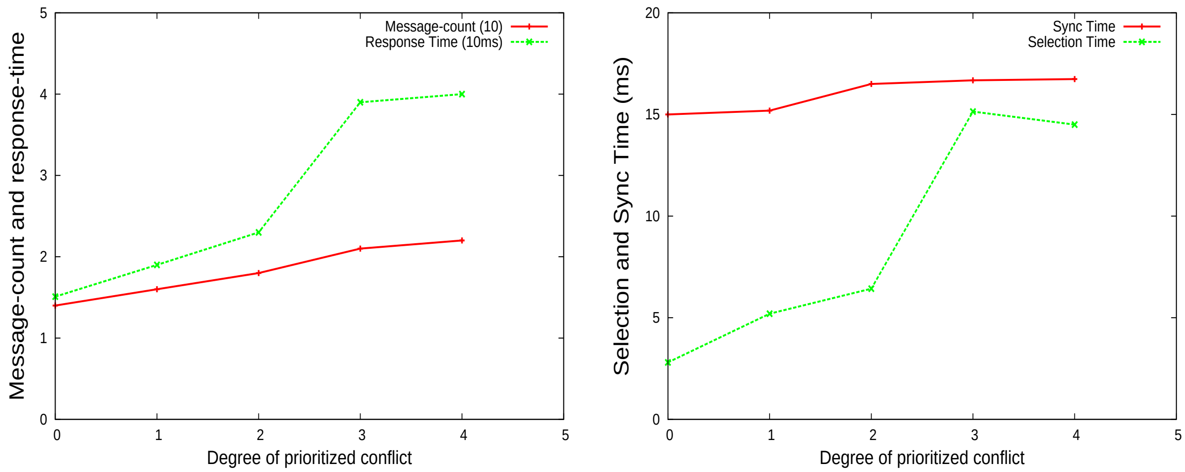

Figure 5: Sensitivity to the degree of *prioritized* conflict

Variation of metric response-time As expected, also the time required to execute the first interaction increases with the degree d of the system, which can also be seen in Figure 5. Again, adding a local conflict (as in the step from $d = 0$ to $d = 1$) leads only to a small increase of the response time as the situation is handled locally. The increase is larger when a global priority is added. Note also that the increase in response time is more important than the increase of the number of messages: up to 20 but indeed, adding a priority requires adding some explicit threads for negotiation, and on the system configuration we use, the time is mainly spent for execution, whereas the communication time is relatively small. Figure 5 shows also the sensitivity of the *sync-time* and the *selection-time* of our prototype to the variation of d. Theoretically, the average synchronization time is independent of the number of conflicting interactions in our system. Indeed, to decide the global readiness of a given interaction, a process has to send and receive a *POSSIBLE* message for this interaction, which is completely independent of whether this interaction is involved or not in a priority rule. This is confirmed by the results of *sync-time* for $d = 0$, $d = 1$ and $d = 2$ (given in Figure 5), for which the synchronization time is almost the same.The synchronization times are slightly greater in case $d = 3$ and $d = 4$. This is due to the order in which messages are received. More precisely, for $d = 2$ priorities are $a_2 < a_1$ and $a_3 < a_1$ which implies that the process negotiating a_3 will send a *READY* message to the negotiator of a_1 to check its readiness. Thus, the negotiator of a_1 may receive and treat this *READY* message before reacting to the *POSSIBLE* messages for the other interaction. We can observe however that for increasing d, the time required to actually choose an enabled interaction, increases considerably. This is not surprising. The fact that the selection time remains relatively small with respect to the synchronization time allows the overall response time increase to remain moderate.

4.1.2 Sensitivity to structural conflicts

Structural conflict arises between interactions when they are all in the *possibleSet* of a common process. To study how our algorithm performs with an increasing number of structural conflicts, we have carried out a series of experiments on a system as depicted in Figure 6. We use a set of systems $T_1, T_2, ..., T_n$ where each T_k has k binary interactions, referred to as a_i $(i = 1, 2, ..., k)$, and $k+1$ processes, referred to as P_i $(i = 1, 2, ..., k+1)$. Processes P_i participate in interaction a_i, and P_{k+1} participates in any interaction. Therefore, all interactions are in *structural* conflict, and the degree of the structural conflict can be

measured by the number of processes in the system.

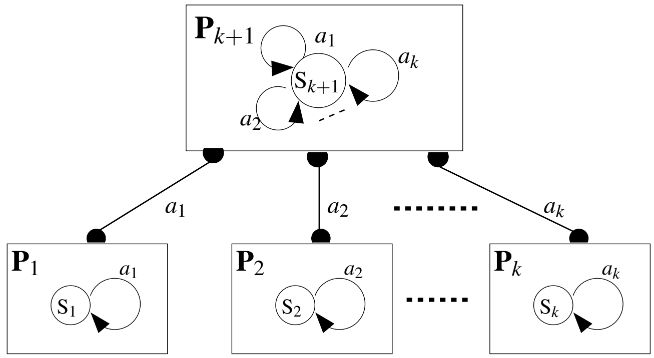

Figure 6: System pattern for experiments (T_k)

Each experiment consisted in executing 100 interactions, and we have evaluated our metrics for up to 5 conflicting interactions (a system with six processes) for several executions of this experiment for each degree of structural conflict.

Variation of message-count We can see in the left side of Figure 7 that our algorithm requires considerably less messages than α-core, where we compare with the numbers provided in [10] for this same example. This is due to the fact that α-core is "connector-centric", that is, it creates an additional process for each interaction whereas our algorithm is process centric, that is all negotiations are hosted by some process and share the same memory space. This means that our algorithm can exploit more local "knowledge" to execute interactions which reduces the number of messages exchanged. When there is no conflict at all (in T_1) both algorithms exchange the same number of messages, then when the degree of conflict increases, our algorithm performs better. The system T_1 has no conflict, and to execute a_1, 3 messages are exchanged (one *POSSIBLE* and two *COMMIT*), thus 300 messages are transmitted during the experiment. When there are conflicts, for T_2 for example, again 3 messages are needed to execute an interaction in the best case, but every time an interaction is refused, at the worst case, a penalty of 3 messages is added (one *POSSIBLE*, one *COMMIT* and one *REFUSE*). To execute 100 interactions, 300 messages are needed in the best case, and 212 extra messages have been added for the situations where an interaction has been refused.

Variation of response-time Figure 7 shows also the selection and elapsed time. Again, the average selection time is in principle independent of the number of interactions in *structural* conflict. Because, when no priorities are added and when an interaction becomes *ready*, only two *COMMIT* messages are exchanged to execute an interaction. Thus the average selection time should be of about $2*\lambda$, where λ is the average message transmission time which in our experimental architecture is $\lambda = 0.2$ ms.

Figure 7 shows that the measured response time is higher. The reason for this is that our implementation is written in Java, and the loop used to send k *POSSIBLE* messages by the process P_{k+1} leads to computational overhead. More precisely, when P_{k+1} enters the loop to send k *POSSIBLE* messages to the different peers, the process P_i which will get the first message sent, will set the interaction i to ready and send back a *COMMIT*. However, P_{k+1} will not treat this message before the termination of this loop. As the actual communication time is low, the *possibleSet* of P_{k+1} may contain many interactions, which increases the *selection-time* (only one interaction is committed, all others must be refused).

4.2 The dining philosophers example

We have carried out a series of tests on the well-known Dining philosophers problem. We consider a variant of the dining philosophers problem inspired from [9] and we propose to deal with this problem

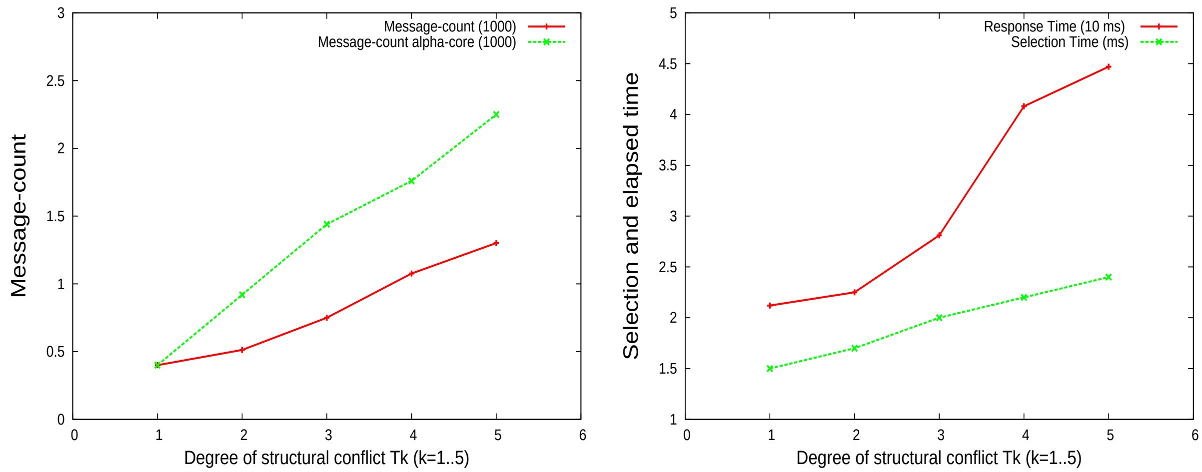

Figure 7: Sensitivity to the degree of *structural* conflict

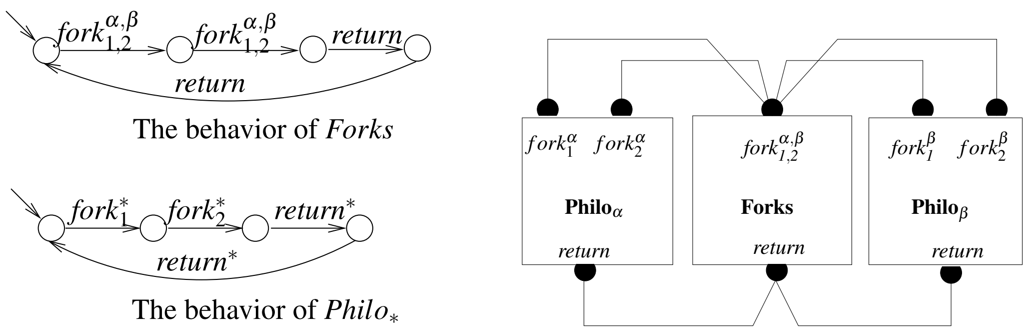

Figure 8: The dining philosophers problem with priorities.

using priorities. Philosophers are seen as processes who provide thoughts if they are given two forks. These forks represent a shared resource. A problem may arise if each philosopher grabs the fork on its right, and then waits for the fork on its left to be released. In this case a deadlock occurs and all philosophers starve.

This deadlock can be avoided by giving higher priority to requests closer to completion. The priority order that is needed here is $\{fork_1^\alpha < fork_2^\beta, fork_1^\beta < fork_2^\alpha\}$. For readability reasons, in Figure 8, the interaction $fork_{1,2}^{\alpha,\beta}$ in the behavior of the process *Forks* corresponds to the interactions $\{fork_1^\alpha, fork_2^\alpha, fork_1^\beta, fork_2^\beta\}$ of the two philosophers. As the process *Forks* participates in all interactions involved in these priorities, *Forks* is designated negotiator for involved interactions and can ensure locally that priorities are respected. Experiments have been carried out for the system with the mentioned priorities (depicted in Figure 8); then we have also considered a system with two philosophers and separate processes for each fork, where deadlock is avoided by the fact that both philosophers first request $Fork_1$, and then $Fork_2$.

Dining philosophers	*Message-count*	*Execution-time*(ms)	$Execution\text{-}time_{Philo^*}$(ms)
With priorities	6	8	30
Without priorities	6	11	45

Table 2: Message count for the dining philosophers

Table 2 shows our measurements for the message-count and the response-time metrics for both systems. We have also measured the average time required for one philosopher to execute a complete cycle (take forks, think and release forks) which we denote by *Execution-Time*$_{Philo^*}$.

We observe that the number of messages exchanged is identical in these two systems. Indeed, priorities are local thus do not induce additional messages. However, using priorities leads to a slight increase of the *execution-time*, as we have already observed in our first example, and the explanation remains unchanged. An additional reason is that the system with priorities has only one process to handle both forks, this making the system less concurrent than the system without priorities. This effect of concurrency is particularly visible in the results for *Execution-Time*$_{Philo^*}$.

5 Conclusion and future work

In this paper, we have presented and evaluated an implementation of the algorithm proposed in [4], which defines a transformation of a global specification of a component-based system with priorities into a distributed system, in which every component becomes a process that may be executed on a different physical machine, and for this purpose is composed with a local controller exchanging messages with peer controllers. We analyze the performance of the algorithm on hand of a number of experiments and measure 3 different metrics by executing the implementation for different systems. These results show that our algorithm behaves as expected. A comparison with the α-core algorithm is performed based on results available in the literature, which shows that our algorithm requires a much smaller amount of messages for systems without priorities. α-core does not handle priorities, and, for the time being, our algorithm does not handle multi-party synchronization which α-core does.

More experimentation is required, and different improvements of our algorithm are envisaged. In particular, beyond the adaptation to multi-party interactions, we plan adapting knowledge-based methods as proposed [12, 2]. This adaptation is not straightforward, as these algorithms are applied at the level of synchronizing processes, thus ignoring how the synchronization are realized in terms of message exchanges, whereas our algorithm addresses exactly this lower level, where additional knowledge may be exploited to decrease the number of messages exchanged without significantly increasing the local memory or the complexity of the local algorithms.

References

[1] Rajive Bagrodia (1989): *Synchronization of Asynchronous Processes in CSP*. ACM Trans. Program. Lang. Syst. 11(4), pp. 585–597. Available at `http://doi.acm.org/10.1145/69558.69561`.

[2] Ananda Basu, Saddek Bensalem, Doron Peled & Joseph Sifakis (2009): *Priority Scheduling of Distributed Systems Based on Model Checking*. In: CAV, Lecture Notes in Computer Science 5643, Springer, pp. 79–93. Available at `http://dx.doi.org/10.1007/978-3-642-02658-4_10`.

[3] Ananda Basu, Marius Bozga & Joseph Sifakis (2006): *Modeling Heterogeneous Real-time Components in BIP*. In: Proc. of SEFM'06, IEEE Computer Society, pp. 3–12. Available at `http://doi.ieeecomputersociety.org/10.1109/SEFM.2006.27`.

[4] Imene Ben-Hafaiedh, Susanne Graf & Sophie Quinton (2010): *Building Distributed Controllers for Systems with Priorities*. Available at `http://www-verimag.imag.fr/Technical-Reports,264.html?lang=en&number=TR-2010-15`.

[5] Simon Bliudze & Joseph Sifakis (2007): *The algebra of connectors: structuring interaction in BIP*. In: Proc. of EMSOFT'07, ACM Press, pp. 11–20. Available at `http://doi.acm.org/10.1145/1289927.1289935`.

[6] K. Mani Chandy & Leslie Lamport (1985): *Distributed snapshots: determining global states of distributed systems*. ACM Trans. Comput. Syst. 3(1), pp. 63–75.

[7] Gregor Gößler & Joseph Sifakis (2005): *Composition for component-based modeling*. Sci. Comput. Program. 55(1-3), pp. 161–183. Available at `http://dx.doi.org/10.1016/j.scico.2004.05.014`.

[8] William Gropp, Ewing Lusk & Anthony Skjellum (1999): *Using MPI (2nd ed.): portable parallel programming with the message-passing interface*. MIT Press, Cambridge, MA, USA.

[9] Luca Padovani (2008): *Contract-Directed Synthesis of Simple Orchestrators*. In: Proc. of CONCUR'08, LNCS 5201, pp. 131–146. Available at `http://dx.doi.org/10.1007/978-3-540-85361-9_13`.

[10] José Antonio Pérez, Rafael Corchuelo & Miguel Toro (2004): *An order-based algorithm for implementing multiparty synchronization*. Concurrency - Practice and Experience 16(12), pp. 1173–1206. Available at `http://dx.doi.org/10.1002/cpe.903`.

[11] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker & J Dongarra (1996): *MPI: The complete reference*. MIT Press, Cambridge, MA.

[12] Doron Peled Susanne Graf & Sophie Quinton (2010): *Achieving Distributed Control Through Model Checking*. In: CAV.

Algorithm 1 Main: **Input:** $possibleSet \neq \emptyset$ **Output:** interaction a

Require: $toNegotiate = \{a \in possibleSet \mid negotiator(a) = P\}$
    Input: set of interactions $possibleSet \neq \emptyset$ **Output:** interaction i
    $prioFree = \{a \in possibleSet \mid \ \nexists b \, . \, b < a\}$
    $waitingSet \longleftarrow \emptyset$
    checking global readiness:
    $notReadySet \longleftarrow \emptyset$
    $readySet \longleftarrow \emptyset$
    $lessPrio(a) = \{b \in readySet \mid b < a\}\}$
    for all $a \in possibleSet$ **do**
        send $POSSIBLE(a)$
    end for
    create WaitingForCommit($possibleSet$)
    if receive $POSSIBLE(a)$ and $a \in toNegotiate$ **then**
        create Negotiate(a) and $readySet \longleftarrow readySet \cup \{a\}$ and
        for all $b \in lessPrio(a)$ **do**
            kill Negotiate(b)
        end for
    end if
    WHEN $\exists$ a $s.t.$ Negotiate(a)= OK or (**receive** $POSSIBLE(a)$ and $a \in prioFree$)
    call TryToCommit(a) and **kill** WaitingForCommit($possibleSet$) and $\forall b \in readySet$ **kill** Negotiate(b)

    if TryToCommit(a)= OK **then**
        return a
    else
        goto checking global readiness
    end if
    if $\forall a \in readySet$ Negotiate(a)= NOK **then**
        goto checking global readiness
    end if
    if receive $REFUSE(b)$ and $b \in readySet$ **then**
        kill Negotiate(b) and $readySet \longleftarrow readySet \setminus \{b\}$
    end if
    if receive $POSSIBLE(b)$ and $b \in possibleSet \setminus \{toNegotiate \cup prioFree\}$ **then**
        send $POSSIBLE(b)$ and $readySet \longleftarrow readySet \cup \{b\}$
    end if
    if receive $NOTPOSSIBLE(b)$ and $b \in possibleSet \setminus prioFree$ **then**
        $notReadySet \longleftarrow notReadySet \cup \{b\}$
    end if
    if receive $POSSIBLE(b)$ and $b \notin possibleSet$ **then**
        send $NOTPOSSIBLE(b)$
    end if

Algorithm 2 Negotiate: **Input:** interaction a **Output:** OK or NOK

Require: $higherPrio(a) = \{c \mid a < c\}$
  for all $b \in higherPrio(a)$ **do**
    send $READY(b)$
  end for
  while $higherPrio(a) \neq \emptyset$ **do**
    if receive $READY(b)$ **then**
      return NOK
    else if receive $NOTREADY(b)$ **then**
      $higherPrio(a) \longleftarrow higherPrio(a) \setminus \{b\}$
    end if
  end while
  return OK

Algorithm 3 WaitingForCommit: **Input:** $possibleSet$ **Output:** interaction a

Require:
  if $waitingSet \neq \emptyset$ **then**
    choose $a \in waitingSet$ and **kill** main and **send** $COMMIT(a)$ and **send** $REFUSE(b)$ for all b in $possibleSet$ and **goto** $Busy(a)$
  else if $waitingSet = \emptyset$ and **receive**$COMMIT(a)$ and $a \in possibleSet \setminus toNegotiate$ **then**
    kill main and **send** $COMMIT(a)$ and **send** $REFUSE(b)$ for all b in $possibleSet$ and **goto** $Busy(a)$
  end if
  if receive $COMMIT(a)$ and $a \notin possibleSet$ **then**
    send $REFUSE(a)$
  end if

Algorithm 4 TryToCommit: **Input:** set of interactions $readySet$ **Output:** OK or NOK

Require:
  send $COMMIT(a)$
  if receive $COMMIT(a)$ **then**
    return OK and **send** $\forall b \in$ readySet$\setminus\{a\}$ $REFUSE(b)$
  else if receive $COMMIT(b)$ and $b \neq a$ and ($b \notin cycle(a)$ or($b \in cycle(a) \wedge P_b = Cyclebreaker$)) **then**
    $waitingSet \longleftarrow waitingSet \cup \{b\}$
  else if receive $COMMIT(b)$ and $b \neq a$ and $b \in cycle(a)$ and $P_b \neq Cyclebreaker$ **then**
    send $REFUSE(b)$ and $readySet \longleftarrow readySet \setminus \{b\}$
  else if receive $REFUSE(a)$ **then**
    return NOK
  end if